1. Preface

Game Summary and Development Philosophy

1.1 Game Summary

SpongeBob UDraw is a large collection of very diverse microgames. We will be employing several different game mechanics – drawing, tilting, dragging (flicking) and tapping/pressing. The game is going to present the player with a rapid succession of microgames in a tightly synchronized manner, so much attention will be paid to asynchronous loading of assets synchronizing music with gameplay.

1.2 Development Philosophy

We believe the best way to approach a task like this is to parallelize as much work as possible, using a substantial number of programmers and allowing each game to be developed in relative isolation from the others such that there are very few common dependencies between microgames.

Our overriding gameplay goal for this project is to prove game mechanics as quickly as possible.

2. Technology Checklist

Following is a list of various tools and technologies we will employ in development of SpongeBob UDraw.

2.1 1st Party Tools & Technologies

Freescale Codewarrior
We will be using Freescale's Codewarrior for Wii as a stand-alone debugger only (MSVC will be our source editing IDE).

NintendoWare for Revolution
We are using NW4R for midi playback, and possibly for authored particle effects.

2.2 3rd Party Tools & Technologies

Adobe Photoshop CS3

Subversion VCS
We will use subversion for both code and data, specifically the Tortoise SVN build. Subversion is very reliable, simple to use, and also free. While there are licensable VCS solutions that have more features, we believe the scope of this game will fit very nicely within subversion.

Microsoft Visual Studio 2008 Standard (or Pro)
We will be using MSVC's IDE rather than the Freescale Codewarrior's IDE for Wii source code development. The Freescale tools will still be leveraged as stand-alone debuggers. We will be employing StudioBlack's BlackBuildVs tool for intergration of the codewarrior compiler/linker/librarian and debugger directly into MSVC.

MS Excel (or optionally OpenOffice Calc)
Used to create localized text assets in spreadsheet form. These tools are used in conjunction with proprietary localization tools described later.

EngineBlack for Wii
We are using StudioBlack's Wii engine and toolset for SpongeBob UDraw Wii.

2.3 New tools & technologies

No new technologies are being developed for SpongeBob UDraw Wii.

3. Development Environment

Tool deployment for each workstation, and project directory structure.

3.1 Tools

We will be using PC's with a mix of XP 32-bit, Windows 7 32-bit, and Windows 7-64 bit. All relevant tools work in each of those environments. Each developer's computer will have the following software installed at minimum, which will allow interaction with the repository and building and running of the game:

- Revolution SDK
- Nintnedoware SDK
- Freescale Codewarrior command line tools.

- Tortoise SVN
- NDEV ODEM software (if an NDEV is to be connected for running the game)
- (Note that Cygwin is not necessary except for machines that must create rvm masters)

Any workstation that will be used to add/edit source files will additionally have Microsoft Visual Studio 2008 Standard or better installed.

We will be using Codewarrior for Wii as a stand-alone debugger only, not to build the game or edit sources.

3.2 Directory structure

The top level directory structure for SpongeBobWii will be as follows:

- SB/Game. The root directory you must synchronize with to build and run the game.

- SB/Game/Assets. All source assets in unconverted form, be they art, text, sound, fonts, etc. The tree may be as deep as necessary, SB/Game/Assets is just the root of the tree. All files under this folder that are necessary to build the game data shall be committed to version control.

- SB/Game/Code. All C++ game sources. This will include any library, SDK, and engine sources as well – everything needed to build the game source.

- SB/Game/Build. All converted data and compiled elf files.

- SB/Game/Build/elfs. The game code will build into this directory. These elfs may be checked into version control, and used by artists to run the current version of the game code against the current version of the assets.

- SB/Game/Build/Intermediate. Temporary and intermediate file storage during the asset building process.

- SB/Game/Build/data. Target build directory for all assets converted by the data build process. This directory will not be version controlled. It may be destroyed and rebuilt automatically.

- SB/Game/bin. All executables needed to build the game code and data.

- SB/Docs. Design docs, charts, diagrams, etc…

4. Core Library Description

A description of EngineBlack's Core library, containing the most basic engine components.

4.1 Introduction

The Core library represents low level hardware and software systems, upon which other EngineBlack libraries and the game are built.

4.2 Memory Management

We will not be using the RVL SDK's MEM library or OSAlloc() functions. We will instead be using an instanceable MemoryManager class. We create 2 top-level instances, one for MEM1 and one for MEM2. Immediately upon startup, the engine takes all of both the MEM1 and MEM2 arenas for this purpose.

All allocated memory blocks returned by a MemoryManager are cache-line aligned to 32 bytes. Allocation can occur from either end of each heap, which helps reduce fragmentation when allocating temporary buffers before more permanent ones. The worst case time complexity for both allocation and free operations is linear with the current number or free blocks.

Sometimes we will create subheaps for special purposes, the memory for which will come from one of the two top-level allocators (for mem1 or mem2). These subheaps may also be managed by an instance of MemoryManager, but more likely will have a more specific allocator class. The Pool<> class is one such custom allocator, which allows O(1) allocation and free operations on buffers of a fixed, predetermined size.

The MemoryManager provides some debug features for tracking memory errors. It can be configured to walk the entire heap list on every memory operation to validate integrity. It can also be configured to intentionally write garbage into all newly allocated memory, and into all freed memory, to immediately flag bugs where

client code assumes the initial value of uninitialized memory, or depends on the continued existence of recently freed memory (both typically very hard to trace bugs otherwise).

The memory manager also tracks the originating file/line of every allocation, even allocations done with 'new'.

4.3 Profiling

While the SDK profiling tools are good for fine-tuning the impact of large numbers of small functions, it does have a few downsides. The finer the testing, the slower your game runs. It requires a devkit to get any output, and requires the programmer to profile “on purpose”, meaning he has to run the profiling tool and set about taking samples of execution.

To augment this, we employ a custom engine profiling module which is always-on, and provides at-a-glance performance metrics.

Key systems in the code that are worth monitoring are wrapped with vp_push() and vp_pop() macros which cause all processing between the macros to appear as a colored bar onscreen, who’s length is proportional to the amount of cycles spent.

As a basic example:

```
vp_push(0xff0000); // red (rgb888)
TickAllObjects();
vp_pop();

vp_push(0xffff00); // yellow
PhysicsSimulation();
vp_pop();

vp_push(0x0000ff); // blue
DrawObjects();
vp_pop();
```

This code would produce three colored bars on screen which immediately tell the observer how long each task is taking relative to each other, and relative to a 60hz or 30hz frame.

The macros can be nested, and have extremely little impact on performance. They may be called tens of thousands of times per

frame with negligible impact. These macros expand to nothing in DEBUG and FINALROM builds.

4.4 Basic Containers

The core library does provide some functionality for intrusive lists, in the form of macros which may be inserted into structures to be listed (links), and structures which contain lists (head/tail pointers). There are also macros for adding/removing from these lists, and for iterating them. These macros are very flexible, and allow any object to be threaded with any number of lists as needed for the design of any system, and are as fast as if the intrusive-list was hand coded each time.

The use of generic, automatically growing containers such as STL is generally discouraged in console game code where the limits of ram are pushed and fragmentation is unacceptable. However, STL containers can be used by some game systems to ease development if will not execute under tight memory requirements (i.e, a casual minigame or the front-end ui).

4.5 Hashing

The core library exclusively relies on a hashing algorithm written by Bob Jenkins. Bob has released his algorithm specifically into the public domain, and so it is free for commercial use.

The core library has adapted JenkinsHash() into several forms, suitable for very fast hashing of u32’s, strings, case-insensitive strings, 2d/3d vectors, and any generic blob of data. It is used by several systems, including the Fonts, Localization, and Asset Management.

4.6 Spatial Management

While not all spatial problems can be efficiently expressed by an abstracted tree or grid class, the core library does provide some spatial classes which can be used where suitable to aid in the acceleration of spatial searches (i.e., the dynamic object visibility system leverages the octree class described next).

Octree/Quadtree
The octree and quadtree classes may be instanced anywhere for any purpose. At construction, the tree is given two limits – max objects and max nodes (cells). New objects and nodes are allocated O(1) from a private Pool<>, and the trees may not be populated beyond these given limits.

One distinguishing characteristic of these trees as opposed to a usual octree or quadree is that these trees are allowed to grow spatially without bound. No extents must be given. New root nodes are added to the top of the tree to accommodate new or moving objects, and existing roots are popped off where possible to re-shrink the space when moving/deleting objects.

To facilitate boundless expansion of the tree, it internally works with two coordinate systems instead of one. The extra one is referred to as "construction space". Construction space is simply an offset from the original space, which is adjusted as roots are added or removed in order to keep construction space from straddling a boundary which demands a higher than necessary root, or an impossible boundary (crossing on any axis).

HashGrid
Provides a boundless grid, sparsely populated with a fixed-maximum number of cells. Cells are accessible in very near O(1) time. Cell coordinates are hashed with JenkinsHash(). Game code can employ this class for situations where it needs very fast proximity testing over a boundless region of similarly sized objects (i.e., which crayons are very close to Fillup).

4.7 Math

Matrix
The core library's matrix class is a template class, generic by number of rows, columns, and data type. This template specification is used to represent both vectors of any size, and matrices as well. This approach allows row vectors and column vectors to be distinct by type. This approach also allows some simple client classes (like the Elastic class described in 5.9) to be generic by rank (number of dimensions).

In practice, only a few specializations are applicable to game development (2x2, 3x3, 4x4, and 1x2, 1x3, 1x4 and their transposes). On wii, only the float versions of those are important. For that reason, each of these are fully specialized for maximum performance.

All matrices are laid out as row major.

Quaternion
Core math provides a quaternion class useful for spherical linear interpolation.

Random
Based on the std library's rand() function, the core library provides several random number generation apis, which simplify the generating of random number for different circumstances.

- Rand(int max) - integer random between 0 and max
- RandRange(int min, int max)
- SRand(int max) - signed integer random between -max and max
- FRand() - a floating point random number between 0 and 1
- FRand(float max)
- FRandRange(float min, float max)
- SFRand() - float random between -1 and 1
- SFRand(float max) - float random between -max and max

Also, an instanceable random number generator is provided (based on a very simple algorithm). It is not as random as rand(), but is very useful for independently running sequences of random numbers that must be reproduceable later from a stored seed.

4.8 Reference Counting

The core library includes a Reference<> template which is very similar to boost::intrusive_ptr. It is is used throughout the engine to provide automatic reference counted ownership for many things, mostly assets (textures, sounds, etc...). References<> may be copied and reassigned freely. They may be released either through destruction or by calling the reset() function of Reference<>.

4.9 Asset Management

All assets in-game are identified by the 32-bit hash of their filename (sans path). So, asset name space is essentially flat.

Assets are built by BlackBuild into one or more contiguous data volume file, such that they can be efficiently loaded whole by the OS. Volumes are the atomic unit of data transaction with the dvd. The game will explicitly mount volumes needed for each part of the game (front end ui, a specific game level). Some volumes may be mounted

once and never unmounted (global data). Volumes are always loaded asynchronously, and may be canceled before their loading completes.

Assets are requested on demand in the runtime using the filename (or hash). If the asset is not already part of a mounted volume, the request will fail and the game will blue-screen with a message indicating which asset was not mounted. Note that during development, assets that are not part of a mounted volume will be loaded from the host computer to satisfy the request. This is a development convenience feature, and allows game prototyping to occur before any thought has to go into organizing game data into volumes. In the final product, all game data will be loaded as part of a small number of large data sets.

4.10 Texture

Texture is the only graphics asset implemented in the Core library – all other graphics assets are implemented by higher level engine libs (i.e., the “Flat” library for 2d images, the “Deep” library for 3d models & materials, and the “Text” library for glyph images). Texture is implemented in Core because it is used by several higher level libraries.

Texture assets are built by the WiiBuildTexture.exe command line converter. All wii texture formats are supported. WiiBuildTexture.exe can accept, tga, bmp, and png files. Tga is generally used as the source image format for textures, although it is often useful to use pngs for this purpose as well.

Textures are reference counted assets. The game makes references to textures in the following way:

```
TextureRef rTexture = Texture::MakeRef(“blah.texture”);
```

TextureRef is a typedef for Reference<Texture>, and insures that the texture survives until all references are destroyed.

**** SpongeBob Udraw note – we will likely make very limited use of Texture assets since this is a 2d game. Use will probably be limited to special effects.*

4.11 Graphics Primitives

The following primitives provided by the core library for simple, custom rendering or for debug graphics.

- Sphere
- Box (filled or outline)
- Cylinder
- Octahedron
- Lines
- Quads (generally the most used)

All primitives have default uv mappings, and may be colored & textured.

4.12 Game tools

Elastic
The core implements a vector motion filter called "Elastic". It is a template which works on any size vector, and can be used to easily govern acceleration/deceleration of any motion in the game. It is particularly useful for UI implementation, camera motion, and many other gameplay applications.

StateVariable
Simplifies the creation of state machines, by putting much commonly needed state-based functionality into one state variable class. This is not a base class for state machines, but a type to use as the current state variable. Keeps track of the last state, whether or not the current state is "new" (first tick in new state), and the number of ticks since the last state change.

Bezier
Basic functions for Bezier curves.

4.13 Home Menu / Strap Reminder

The home menu is implemented in the wii version of the core library. It imposes no architectural constraints on the game at all, and very few areas of the game code need to be aware of the home menu's existence. The only game code that must be aware of the home menu is code that must disable either the menu or the icon (very few places). Otherwise, the home menu is transparent and lotcheck compliant

There is also a core StrapReminder module which allows a game to implement the strap reminder screen quickly in a lotcheck compliant way.

4.14 Screen Shots

The Core library transparently implements the Nintendo Provided screenshot.c code. All builds of the game will work with Nintendo's screens hot utility except for a GMC build. In GMC builds, this and other development features are automatically disabled.

5. Render Library

Description of EngineBlack high-level render library.

5.1 Camera

A simple engine-level camera class which game cameras may contain or derive. This is a freely instanceable class, and the “current camera” from the engine’s perspective just means the camera that is passed to the top level scene drawing functions.

There are both 2d and 3d Cameras, both very similar in implementation and interface. The camera maintains the following state:

- World basis vectors and position vector of the camera
- View transform (inverse of world basis & pos)
- Projection transform (for 3d may be ortho or perspective)
- Inverse projection transform
- Pre-multiplied view * projection matrix
- Pre-multiplied inverseProjection * cameraWorld mtx (inverse view)
- Clipping planes (2 slabs for 2d, 1 slab and 4 planes for 3d)

The camera does not contain any camera-behavior code, it only exists to express a view volume. All properties maintained by the camera are derived mathematically from a just a projection matrix and a camera world matrix.

The core camera also provides mappings between world and eye/viewport coordinates. For instance, a 2d screen position for an

on-screen pointer can be transformed by the 3d camera into a world-space ray which passes through all points that project back to the input screen coordinates.

5.2 Other Renderer Library Features

All other Render library functions are related to 3d rendering are not applicable to SpongeBob Udraw.

6. 2D Rendering Library (“Flat”)

“Flat” is the name of EngineBlack's 2d rendering library

6.1 Images

Provides display of static images. All wii texture formats are supported. Images are authored as either bmp, tga, or png under the Assets directory (png is preferred). In addition to the image pixels, Images have an origin point which is specified by an 'attr' file next to the image file itself (i.e. Button.png and Button.png.attr). If no attr file exists for an image, the origin point defaults to the center of the image.

The game loads images by making references to image filenames as follows:

```
Flat::ImageRef rImage = Flat::Image::MakeRef(“blah.image”);
```

ImageRef is a typedef for Reference<Image> (Reference<> is described in the Core library section). The use of a MakeRef() function is common to many other EngineBlack asset types.

Images are created using the WiiBuildImage.exe command line converter.

6.2 Anims

An anim is a single file containing multiple images that are to be displayed in sequence. Anims are divided into clips, which are sequences of individual images. Anims are not built on top of the above described Image asset, but rather are their own separate asset.

Clips are defined by placing a .clip file somwhere under the assets directory. A .clip file causes all png's at its directory level and below to become included in the clip. Clips are likewire included into Anim assets by a .anim file placed somewhere at or above the directory level where the clip(s) exist.

Anims do not redundantly store identical image frames. These are canceled out during conversion. Anims are converted by the WiiBuildAnim.exe command line converter.

The game loads anims by making references to anim filenames as follows:

```
Flat::AnimRef rAnim = Flat::Anim::MakeRef("blah.anim");
```

A loaded anim file may be queried for it's individual clips and frames. The Flat library does not actually contain functionality to playback an animation – only to define and display them. Animation playback is described in the Component library section.

6.3 Other Flat Library Features

The Flat library contains tile-based image and image compression features which are not applicable to SpongeBob Udraw.

7. Text Library

A description of EngineBlack's text rendering and localization tools.

7.1 Text and Localization

All game text throughout the engine is handled as utf-8. This allows us to express any character from the full Unicode definition, yet still handle text at runtime with the familiar strcpy/strcmp family of functions. It also enables us to easily intermix standard 7-bit ASCII strings during development and debugging.

Text can be expressed as text asset files, which are created offline by converting specially formatted MS Excel or Open Office Calc

spreadsheets. Text which does not need localization may also be expressed as hard coded ASCII literals, or loaded from a ASCII or utf-8 raw text file.

If localization spreadsheets are used, text strings may be retrieved at runtime from a text asset using the id of the string, and the id of the localization. Both the string id and the localization id are case-insensitive 32-bit hashes of the corresponding name (i.e. “English”, “French”, “GameOver_Message”). Text assets are essentially a two dimensional hash table, with string IDs on one axis and localizations on the other, mirroring the spreadsheet they are authored with.

7.2 Fonts

Fonts in EngineBlack are defined by ttf or otf files, rendered by the FreeType open source library. We use FreeType to offline render bitmap versions of the font glyphs, rather than linking freetype into the game's runtime. This approach avoids potential legal issues with distributing verbatim otf or ttf files on disc, or distributing code that includes potentially proprietary algorithms to render those glyphs at their best quality. It also produces smaller files in virtually all cases, as monochrome glyph bitmaps rle extremely well.

Fonts are defined by placing a .font file somewhere under the source Assets directory. Font files are small xml files which specify the following attributes of an individual font:

- the ttf or otf file that contains the glyph descriptions
- the pixel size at which to render the font
- the unicode characters to sample, expressed in one of several forms:
    - an explicit string of characters written into the font xml file
    - one or more text files who's characters should be included
    - one or more localization spreadsheets who's characters should be included

Fonts are converted using the BuildFont.exe command line converter. Fonts are automatically rebuilt anytime one of the text sources used to determine codeset changes.

In the runtime, a client of the font system will make references to glyphs it needs to draw. And actively referenced glyph will be converted in software to the Wii’s hardware texture format when the first reference is made. When all references to a glyph have expired, the hardware texture image is destroyed. Note that glyphs are always

expressed by their own texture, they are never packed onto a larger texture.

Alternatively glyph images from a font may be used by a software text renderer, rather than converting glyphs into hardware textures. In this case, referencing glyphs is not necessary – the images are used by the software text renderer straight from the font asset with no conversion required.

7.3 Other features

Formatting and markup
The Text library provides a formatting system capable of using simple markup to mix different typefaces and sizes into the same string of text. It also can format a block of marked-up text into multiple "fragments" which can be drawn individually by the client using different colors, animation, or other effects.

Stroke
Text drawn by the font library can be stroked. Multiple strokes can be applied ot the same text if desired.

8. Transition Library

The Transition library is a common hook for various screen fades and wipes that occur between gamestate changes.

9. Sound Library

The EngineBlack sound library is a simple wave playback library built on the wii's AX library. Waves are converted from aif or wav files using WiiBuildWave.exe. Loop points are supported.

Waves are reference counted assets. The game makes references to waves in the following way:

```
WavRef rWave = Wave::MakeRef("blah.wave");
```

The runtime provides an api for playing back waves. A played wave produces a SoundHandle that can be used by client code to monitor the playback progress and change playback parameters in real time (i.e. volume, pitch, pan). SoundHandles are globally unique, the same handle will not be returned twice. In this way, once a sound handle is obtained, it never expires.

10. Component Library

This EngineBlack library provides low-level game programming components that can be included into game objects and scenes.

10.1 Process

The Process / ProcessManager class pair provide synchronous ticking functionality to game objects and scenes. Any game class may contain (not inherit) one or more Process members. Each member process is pointed a tick function (a member function of the game class) using a efficient delegate mechanism provided in the core library. Instances of that game class will by “ticked”, once for each process it contains.

Similarly, game scene classes will include one or more ProcessManager classes. Process instances from game objects are added to the scene's process manager (usually there is just one process manager per scene).

10.2 VisibleSpace2 / VisibleObject2

The VisibleSpace2 / VisibleObject2 class pair provide 2d drawing functionality to game objects. Game objects may contain one or more VisibleObject2 instances, each with its own world transformation. Similarly, the game scene will contain one or more VisibleSpace2 instances to which all VisibleObject2's will be added.

For each game frame, the VisibleSpace2 instance(s) will be drawn using a Camera2 from the Render library as the view volume. VisibleSpace will cull and sort its member objects call the draw function of each object which is visible.

VisibleObject2 is usually not used directly by the game. The Component library provides specialized versions of VisibleObject2 for

display Flat::Image and Flat::Anim assets. The game can directly use a VisibleObject2 to insert it's own low level drawing into a 2d scene.

VisibleObject2 and VisibleSpace2 also provide transparency and sorting functionality.

10.3 UI Component library

A ui specific version of the above components exist as the foundation for building ui’s. One key difference is that the ui visible objects are hierarchical.

10.4 Other Component Library classes

Other component library functionality is not applicable to SpongeBob udraw.

11. SpongeBob Udraw Architecture

This section describes implementation details specific to SpngeBob udraw.

11.1 Framerate

The game will run at 60hz at all times.

11.2Memory Maps

Although the uma architecture of the Wii allows all types of data to reside in either MEM1 or MEM2, we will generally divide the memories in the following way:

- MEM1 (24 mbyte). Generally dedicated to executable code, and any game data that is frequently read by the CPU (i.e. collision, visibility, AI, other game logic data)
- MEM2 (64 mbyte). Generally dedicated to large sound or gfx assets which do not require frequent CPU reads, or any reads at

all. Also any runtime-created textures, such as font glyph images will go here.

The purpose behind this strategy is that MEM2 has a much slower CPU read speed than MEM1, although the CPU may write to both memories equally fast. MEM2 will essentially be treated as vram and audio ram, with the added bonus that the cpu can access it directly.

11.3 Disc Usage

With the possible exception of some front-end ui music, we will not be streaming audio from the disc. We will absolutely not be streaming audio during gameplay. Disc bandwidth during gameplay will be reserved for asynchronously loading assets for the next microgame. Any fmv will not be played between synchronized microgames.

Data packages as described in Core library section will be the atomic unit of resource loading and unloading. Data packages are designed to be loaded very fast. We will layout the disc such that microgame packages reside at the outer edge such that they will load faster.

We anticipate that the dvd capacity will dwarf our sum total of assets and fmvs.

11.4Visibility & Camera

While SpongeBob UDraw will use the Render libraries 2d camera, we are unlikely to ever move the camera from the default position for any of the games (except perhaps for camera shake effects). Also, while we will be using the VisibleSpace / VisibleObject system from the Component library, we will disable the normal frustum culling function as we will not have any large off-screen portions in any microgame.

11.5 Collision

The collision needs of the microgames mostly range from none at all to extremely simple, such that we will not be using a generalized collision library. There are some game designs that do some simple sphere vs aabb and line vs line collision, and these will be coded directly into those few games.

11.6 Scene and Entity Management

All games will be built upon an extremely thin MicrogameWorld and MicrogameObject class pair. This class pair is so thin in fact that it does even include object positions. MicrogameWorld will contain only the process manager and visible space for the games (see the section on the Component library), and parameters for the current game (speed, variation, etc…). Games will include common components like Process and VisibleObject directly in their own object classes, rather than inheriting for them from an Entity class common to all games. We feel this approach is essential to maintaining such large number of diverse games.

11.7 Sound

All sound effects will be triggered using a sound-specific event system developed by Wayforward. The game code will ultimately not play wavs directly, but will invoke event trigger files (called sfxtr files) which are created by the audio engineer. This gives the audio engineer the flexibility to modify a wide range of playback parameters per sound effect without changing game code.

The invocation of sound effects will all be done in game code. For SpongeBob wii, there will be no data-driven way to author when specific sound effects are invoked.

11.8 Common game objects

Canvas

Several minigames will require painting and drawing. The game code will contain a Canvas class with a simple api for plotting lines with brushes and for displaying the canvas. This will form the visual basis of all our drawing mechanic games. The Canvas class will store the image in the wii's 32-bit texel format.

Pointer

Several minigames also will need a pointer, which will be implemented as a common game object utilizing the Process and VisibleObject components from the Component library.

Timer

All games will implement a timer, which will be synchronized to midi music playback.

11.9 Synchronization with MIDI music

The duration of all microgames is slaved to the game music (8 beats per game). The music speeds up or slows down with difficulty. Given these design requirements we will use midi for the game music.

We will not use midi cue markers for synchronization. Instead, we will time the music playback using the wii's high resolution timer. This is preferred because all sequences to be synchronized with are either 4 or 8 beats in length.

11.10 The Microgame Queue

A GameQueue class will be implemented to handle the complexities of scheduling games to be played in advance, loading subsequent game data while the current game is being played, and insuring that changes between games and interstitial screens happens seamlessly with no hiccups in the music playback.